(1)

Computer Corporation of America

575 Technology Square
Cambridge
Massachusetts 02139
617-491-3670

(1)

QUERY PROCESSING IN SDD-1: A SYSTEM FOR DISTRIBUTED DATABASES

Nathan Goodman
Philip A. Bernstein
Eugene Wong
Christopher L. Reeve
James B. Rothnie

October 1, 1979

Computer Corporation of America
575 Technology Square
Cambridge, Massachusetts 02139

This research was supported by the Advanced Research Projects Agency of the Department of Defense under Contract No. N00039-77-C-0074, ARPA Order No. 3175-6. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Advanced Research Projects Agency or the U.S. Government.

Abstract

This paper describes the techniques used to optimize relational queries in the SDD-1 distributed database system. Queries are submitted to SDD-1 in a high-level procedural language called Datalanguage. Optimization begins by translating each Datalanguage query into a relational calculus form called an envelope, which is essentially an aggregate-free QUEL query. This paper is primarily concerned with the optimization of envelopes.

Envelopes are processed in two phases. The first phase executes relational operations at various sites of the distributed database in order to delimit a subset of the database that contains all data relevant to the envelope. This subset is called a reduction of the database. The second phase transmits the reduction to one designated site, and the query is executed locally at that site.

The critical optimization problem is to perform the reduction phase efficiently. Success depends on designing a good repertoire of operators to use during this phase, and an effective algorithm for deciding which of these operators to use in processing a given envelope against a given database. The principal reduction operator that we employ is called semi-join. In this paper we define the semi-join operator, explain why semi-join is an effective reduction operator, and present an algorithm that constructs a cost effective program of semi-joins given an envelope and a database.

Table of Contents

Table of Figures

1. Introduction

SDD-1 is a prototype distributed database management system being developed by Computer Corporation of America. SDD-1 permits a database to be distributed among the sites of a computer network, yet accessed as if it were stored at a single site. Users interact with SDD-1 by submitting transactions written in a high-level procedural language called Datalanguage [CCA]. Query processing in SDD-1 amounts to translating each transaction into a sequence of commands that access data at local sites and move data between sites to perform the transaction's computation. This translation is the subject of this paper. Other aspects of SDD-1 are presented in [BSR, HS, RBFG].

The SDD-1 system architecture is described in [RBFG]. For purposes of this paper a simplified model will suffice. The system consists of a collection of sites fully connected by a communication network. Each site is a full-scale computer (as opposed to a micro-computer) and manages a portion of the database using a local database management system (abbr. DBMS). The database and each local DBMS are assumed to be relational; a review of

relational terminology appears in Figure 1.1. The network is logically a point-to-point network (i.e., it does not support point-to-multipoint broadcast), and is assumed to have Arpanet-like performance characteristics[1].

The critical query processing problem in this environment is one of query optimization. Sustainable bandwidth on Arpanet is at most 10,000 bits per second; this is some three orders of magnitude lower than transfer rates between disk and main memory in typical full-scale computers. As a consequence, processing strategies with good performance in a centralized DBMS can easily explode in a distributed environment, running hundreds of times more slowly. Our principle objective is to avoid this performance degradation.

Stating our query optimization problem more precisely, we are given a transaction T and a database D which is statically distributed without replication[2]; our goal is to compute T(D) with a minimum quantity of inter-site data transfer. That is, we assume network bandwidth to be the system bottleneck, and our optimization objective is to minimize use of this resource. Other resources, notably

---

1. SDD-1 is implemented on Arpanet.
2. SDD-1's handling of replicated data is discussed in Section 5.

Review of the Relational Data Model Figure 1.1

(a) Relational Data Objects

Term	Definition
domain	a set of values
attribute	an alternate name for a domain
relation schema	a description of a relation, consisting of a relation name and list of attributes
relation	a subset of the cartesian product of the domains of the attributes of the corresponding relation schema
tuple	an element (or row) of a relation
database	a set of relations

(b) Relational Algebraic Operations

Selection: $R[A=x] = \{r \in R \mid r.A=x\}$
where r.A is the value of the A-domain in tuple r

Projection: $R[A_1,A_2,\ldots,A_n] = \{<r.A_1,r.A_2,\ldots,r.A_n> \mid r \in R\}$

Join: $R[A=B]S = \{rs \mid r \in R, s \in S, \text{ and } r.A = s.B\}$

---

local DBMS computation, are assumed to be free; in practice, local DBMS activity would be optimized as a secondary objective, but this issue will not be considered here.

Other cost factors we ignore include distance effects, the effects of network loading, and the overhead costs incurred whenever sites interact. We believe the first two effects to have second-order importance only. The

third factor has much greater importance and precludes query processing strategies that employ large numbers of interactions [RGM]. Although we do not consider this factor explicitly, it is taken into account by the structure of the processing strategies we consider. As the reader will see, we always translate transactions into programs with relatively few interactions.

Our solution has two main steps. The first step translates the user's Datalanguage transaction into an internal QUEL-like form [HSW]. All aspects of query processing that depend on Datalanguage are handled in this first step. The second step optimizes the processing of the internal form. This step is quite general and can be used without modification in other distributed database systems. This paper emphasizes the optimization techniques of step two which we consider to be our principal contribution.

The paper is organized in six sections. Section 2 develops our paradigm for transaction execution, and defines the internal form that we subsequently optimize. Sections 3 and 4 describe the optimization of this internal form: Section 3 defines the "solution space" of the optimization -- i.e., the types of operations available for processing the internal form; and Section 4

presents our optimization algorithm for producing efficient sequences of these operations. The mapping from Datalanguage to internal form is explained in Section 5. Section 6 summarizes our technique and suggests extensions.

An early version of the SDD-1 query processing algorithm is described in [Wong]. Other approaches to distributed query processing appear in [ESW, HY, Willcox].

2. Query Processing Paradigm

Perhaps the simplest strategy for processing a transaction T against a distributed database is to move all relations referenced by T to a single site, and then execute T at that site. The disadvantage of this strategy is that it incurs unacceptably high communication cost. Our query processing paradigm is to perturb this simple strategy into an efficient one by using relational operations to reduce the size of each relation before moving it.

Distributed query optimization in our paradigm is concerned with performing this "reduction" process correctly and efficiently.

2.1 Reduction

Database state $D'=\{R'_1,\ldots,R'_n\}$ is a sub-state of $D=\{R_1,\ldots,R_n\}$ if R'_i can be obtained from R_i by selection and projection operations, for $i=1,\ldots,n$. A reduction of database state D relative to transaction T is any sub-state D' such that $T(D')=T(D)$. Intuitively, a

reduction eliminates portions of the database that are irrelevant to T. In general, many reductions exist for each T and D. Given T and D our optimization task is to compile T into a program RHO such that

a. RHO(D) is a reduction of D relative to T,
b. all relations in RHO(D) are present at a single site, and
c. RHO incurs minimum cost (when applied to D) over all programs satisfying (a) and (b).

If RHO satisfies (a) and (b) for all D, then RHO is called a reducer for T.

2.2 Envelopes

To construct the desired program RHO, we find it necessary to analyze the body of T. However, Datalanguage transactions are approximately as general as programs written in a high-level programming language and it is difficult to analyze them directly. Therefore, we map each Datalanguage transaction into a QUEL-like internal form called an envelope, and optimize the envelope instead of the transaction. The mapping from transaction to envelope is dependent, of course, on details of

Datalanguage, and hence is specific to SDD-1; this transformation is described in Section 5. Having obtained an envelope, however, the remainder of our technique is applicable to other distributed relational DBMSs.

Syntactically, an envelope is essentially a QUEL query. An envelope, $E_{q,t}$, consists of a qualification, q, and a target list, t. A qualification is a boolean combination of selection clauses of the form (R.A=constant) and join clauses of the form (R.A=S.B), where R.A and S.B are indexed-variables and denote attribute A of relation R and attribute B of relation S respectively[3]. We assume that qualifications are pure conjunctions; disjunction is handled by placing the qualification in disjunctive normal form and treating each conjunction separately. A target list is a set of indexed-variables.

The result of applying $E_{q,t}$ to database D is defined by the following procedure:

---

3. Note that we avoid tuple variables. Tuple variables can be accommodated by (conceptually) duplicating a relation and thereby having two relation-names range over it. We also avoid more general clauses, e.g., R.A<S.B, for pedagogic simplicity. They can added without altering the technical claims that follow.

1. Solve $E_{q,t}(D)$ as a query, using QUEL semantics; i.e.

   a. construct the cartesian product of the relations in D;
   b. eliminate tuples from the cartesian product that fail to satisfy qualification q; and
   c. project the remaining cartesian product onto the target-list t.

2. For each relation, R, project the result of (1) onto the attributes of R referenced in t, thereby producing a sub-state of D.

E is an envelope for T if for all database states D, T(E(D))=T(D), i.e. E(D) is a reduction of D relative to T. Intuitively, an envelope for T "envelopes" or delimits the portions of the database needed to process T. In general there are many envelopes for a given transaction; a good envelope is one that tightly delimits the data needed by T. Finding good envelopes is an optimization problem that depends on the language for expressing transactions. The solution used by SDD-1 appears in Section 5, but a general solution is not attempted. Figures 2.1-2.3 illustrate a database, a Datalanguage transaction, and an envelope for it.

Example Database — Figure 2.1

Relation Schema	Location of the Relation
SUPPLIER(S#, NAME, STREET, CITY, STATE)	site 1
SUPPLY(S#, P#, QTY, PRICE)	site 2
PART(P#, FUNCTION, SPEED, PACKAGE)	site 3

Example Transaction T_1 — Figure 2.2

Note: Datalanguage constructs used in this example are explained in Section 5.

Description of transaction:
For each 7401-equivalent part supplied by a Massachusetts supplier, print the supplier number, name, address, price, and part number. In addition, print how many of these parts have switching speeds of 2 nano-seconds.

```
Transaction T1:
  Begin
    Count:=0;
    For SUPPLIER
      If SUPPLIER.STATE="MA"
      Then For SUPPLY
           If SUPPLIER.S#=SUPPLY.S#
           Then For PART
                If SUPPLY.P#=PART.P# and PART.FUNCTION=7401
                Then Begin
                     Print SUPPLIER.S#, SUPPLIER.NAME,
                           SUPPLIER.STREET, SUPPLIER.CITY,
                           SUPPLIER.STATE, SUPPLY.PRICE,
                           PART.P#;
                     If PART.SPEED=2
                     Then COUNT:=COUNT+1;
                     End
      Print "Number of 2-nanosecond versions is", COUNT;
  End
```

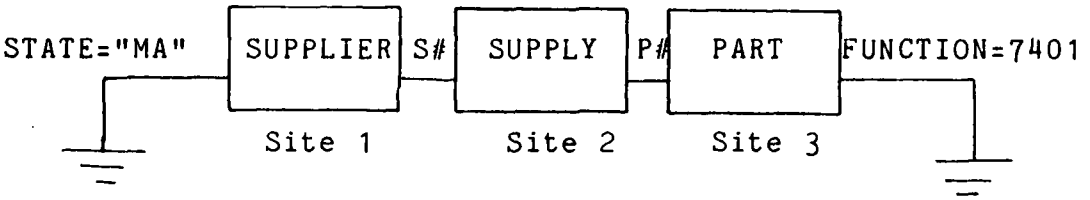
Example Envelope E_1, for T_1 Figure 2.3

Envelope E_1:

target-list:{SUPPLIER.S#, SUPPLIER.NAME, SUPPLIER.STREET,
SUPPLIER.CITY, SUPPLIER.STATE, SUPPLY.S#,
SUPPLY.P#, SUPPLY.PRICE, PART.P#,
PART.FUNCTION, PART.SPEED}

qualification: SUPPLIER.STATE="MA"
and SUPPLIER.S#=SUPPLY.S#
and SUPPLY.P#=PART.P#
and PART.FUNCTION=7401

graph representation of the envelope:



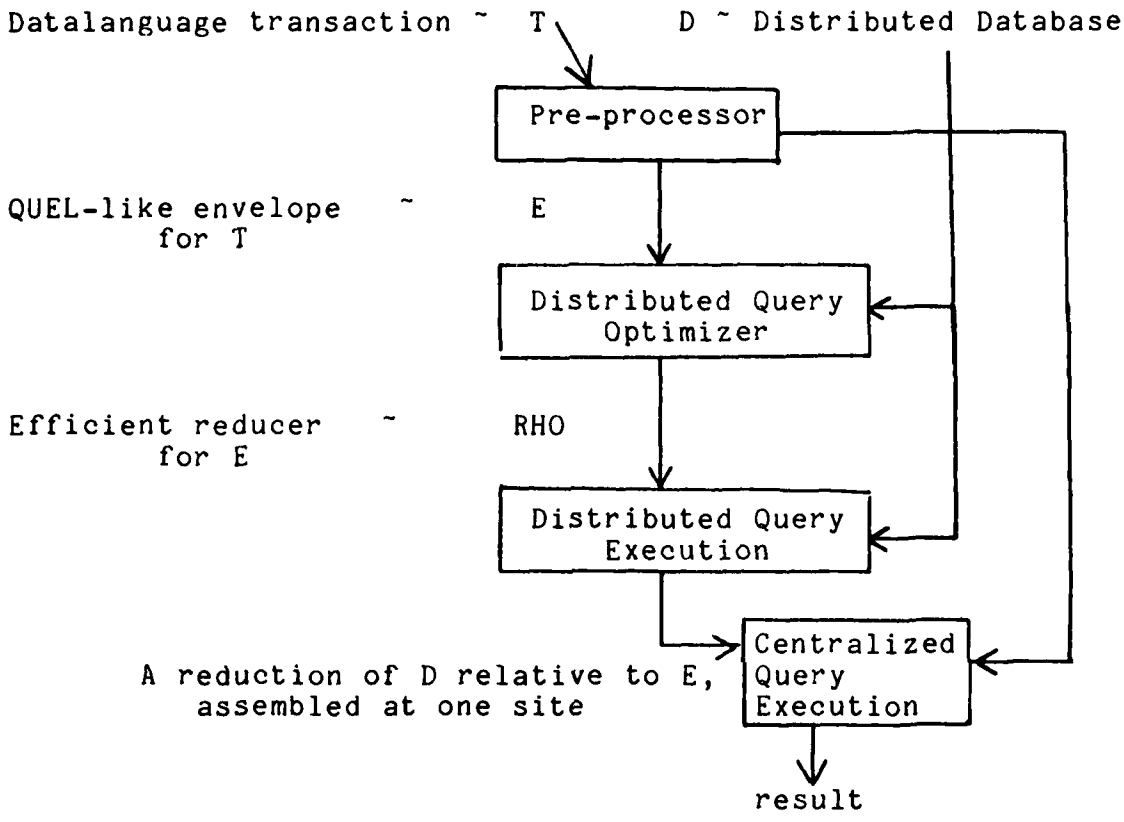
Importantly, if E is an envelope for T, then every reduction of a state D relative to E is also a reduction relative to T (i.e., E(D')=E(D) implies T(D')=T(D)). By way of proof, let D' be a reduction of D relative to E; so E(D')=E(D) by definition of reduction, and T(E(D'))=T(E(D)). Since T(E(D))=T(D) by definition of envelope, we have T(E(D'))=T(D) as desired. Consequently, every reducer for E is also a reducer for T.

Update transactions are also handled by this paradigm. Suppose U is an update transaction and E is an envelope for U (i.e. U(E(D))=U(D)). E is processed exactly as in the retrieval case: a reduction relative to E is assembled

at one site, and U is executed on that reduction at that site. The result is a temporary file that lists all data items modified by U and their new values. These modifications are propagated to sites holding copies of those data items, and are installed using techniques described in [BSR].

Thus we have mapped the problem of finding efficient reducers for Datalanguage transactions into the problem of finding efficient reducers for envelopes. The next two sections address this latter problem. This paradigm is outlined in Figure 2.4.

Query Processing Paradigm
Figure 2.4



3. Reduction Operations

We model a reducer RHO as a sequential program[4] containing reducing statements and assembly statements. Reducing statements apply relational operations to the database to compute the desired reduction; assembly statements move the resulting reduction to an assembly site where the original transaction is subsequently executed. This section describes the operations used by reducing statements.

3.1 Reduction Tactics

An operation is called legal for E if it maps any reduction relative to E into another such reduction. The purpose of this subsection is to characterize the set of legal operations for E, denoted OMEGA(E).

---

4. RHO is, however, executed in a way that exploits parallel processing. See Section 3.3, and [RBFG].

3.1.1 Projections and Selections

The set of legal projections for E is

{R[X] ¦ R is a relation referenced by E, and X is the set of all attributes of R referenced in E}.

The set of legal selections for E includes

{R[A=k] ¦ R.A=k is a clause of E},

although additional legal selections may be implied by transitivity (see Section 3.1.2). The legality of both sets of operations is obvious.

For example, given envelope E_1 of Figure 2.3, the following operations are legal, and can be used to reduce the database:

1. SUPPLIER[STATE="MA"],
2. SUPPLY[S#,P#,PRICE],
3. PART[P#,FUNCTION,SPEED], and
4. PART[FUNCTION = 7401].

Projections and selections have zero cost under the assumptions of Section 1, since they require no inter-site data transfer. Consequently, every legal one should be included in every reducer.

3.1.2 Semi-Joins

Semi-join is a relational operation that exploits the join clauses of E for reduction purposes.

A semi-join is "half" of a join; the semi-join of relation R by relation S on attribute A, denoted R<A=A]S, is defined to be $(R[A=A]S)[ATT_R]$, where ATT_R denotes the attributes of R. Equivalently,

$$R<A=A]S = \{r \in R \mid (\exists s \in S)(r.A=s.A)\}.$$

Intuitively, R<A=A]S eliminates every tuple of R that fails to join with any tuple of S.

Since R<A=A]S = R<A=A](S[A]) = R[A=A](S[A]), not all of S is needed to compute R<A=A]S; only S[A] is required. Thus if R and S are stored at different sites, R<A=A]S can be computed by transmitting S[A] to R's site; it is not necessary to ship all of S.

Semi-joins have several important properties that make them valuable.

i. $R<A=A]S \subseteq R$

ii. $R[A=A]S = (R<A=A]S)[A=A]S$

iii. $R[A=A]S = R[A=A](S<A=A]R)$ [5]

By (i), a semi-join can only decrease (never increase) the size of its left operand. By (ii) and (iii), a preliminary semi-join does not alter the result of a later join on the same clause. It follows that the set of legal semi-joins for E includes

$\{R<A=A]S, S<A=A]R \mid R.A=S.A$ is a clause of E$\}$.

As with selections, additional legal semi-joins may be implied by transitivity.

The set of operations implied by transitivity is obtained by constructing a node- and edge-labelled undirected graph G_E whose nodes are the indexed-variables and constants of E, and whose edges are

$\{\{N_i,N_j\} \mid N_i=N_j$ is a clause of E$\}$.

Then we construct the transitive closure of G_E, denoted G_E^+; G_E^+ is a graph with the same nodes as G_E, but whose edges are

$\{\{N_i,N_j\} \mid N_i$ and N_j are connected by a path in $G_E\}$.

G_E^+ can be computed efficiently using [Algorithm 5.2, AHU]. Given G_E^+, the set of legal selections for E is

$\{R[A=k] \mid \{R.A,k\}$ is an edge of $G_E^+\}$,

---

5. Unlike join, semi-join is not symmetric, hence $R<A=A]S \neq S<A=A]R$. The former reduces R, while the latter reduces S.

and the set of legal semi-joins for E is

$\{R\langle A=A\rangle S,\ S\langle A=A\rangle R \mid \{R.A,S.A\}$ is an edge of $G_E^+\}$.

(Proofs appear in [BC,BG].)

Unlike selections, semi-joins generally have non-zero cost, and not all legal semi-joins are necessarily profitable. For example, the following semi-joins are both legal for envelope E_1:

1. SUPPLIER<S#=S#]SUPPLY, and
2. SUPPLY<P#=P#]PART,

If the database state has the characteristics shown in Figure 3.1, then the cost of the first semi-join equals the "size-of" SUPPLY[S#], which equals its width (i.e., the size of each tuple) multiplied by its cardinality (i.e., the number of tuples), which equals 1000. The benefit of the semi-join equals the amount by which it reduces SUPPLIER, which equals the size-of SUPPLIER minus the size-of SUPPLIER<S#=S#]SUPPLY; this benefit is at least 13*4000 = 52000, since at most 1000 SUPPLIER tuples can survive the semi-join. Thus, this semi-join is profitable. However, assuming SUPPLY[P#] $\subseteq$ PART[P#], the second semi-join is not profitable, since it does not reduce SUPPLY at all. Techniques for estimating costs and benefits of semi-joins are presented in Section 3.2.

Profile of Database from Figure 2.1 — Figure 3.1

	SUPPLIER(S#,	NAME,	STREET,	CITY,	STATE)
cardinality	5000 5000	-	-	-	50
width	13 1	3	3	3	3

	SUPPLY(S#,	P#,	QTY,	PRICE)
cardinality	100000 1000	10000	-	-
width	4 1	1	1	1

	PART(P#,	FUNCTION,	SPEED,	PACKAGE)
cardinality	10000 10000	200	-	
width	6 1	1	1	3

cardinality(domain(S#)) = 5000
cardinality(domain(P#)) = 10000

Legend:

cardinality = number of distinct values in a relation column, or an underlying domain.

width = number of bits, bytes, etc. per tuple, or column; widths are given in arbitrary units, with numeric fields having width 1 and string fields width 3.

blank entries are not relevant for the examples discussed in this paper.

Profiles are explained further in Section 3.2.

3.1.3 Join

Join is another operation that can potentially be used to reduce a database. We choose not to use join for this purpose, however, because the "reductive effect" of any single join can be obtained by using two semi-joins, usually at lower cost.

Let RS=R[A=A]S be an arbitrary join. Since our goal is to reduce the database, the relevant effect of this operation is its reductive effect on R and S; this effect is $RS[ATT_R]$ and $RS[ATT_S]$, where ATT_R and ATT_S denote the attributes of R and S respectively. Notice that

$RS[ATT_S]$ = {s | <r,s> ∈ RS}, by definition a projection

= {s∈S | (∃r∈R)(r.A=s.A)},

by definition of R[A=A]S

= S<A=A]R, by definition of semi-join.

Thus the reductive effect of R[A=A]S on S can be attained by the semi-join S<A=A]R; by a similar argument, the effect on R can be attained by R<A=A]S.

Now let us compare the cost of the one join to that of the two semi-joins. To compute R[A=A]S, one of the relations, R say, must be shipped to the other's site.[6] Under the

---

6. Techniques such as query feedback [Rothnie] may be able to decrease the quantity of data shipped, but intrcduce excessive inter-site interactions [RGM].

assumptions of Section 1, the cost of this operation equals the size-of R. To compute the semi-join, we ship R[A] to S and S[A] to R, for a cost of size-of R[A] + size-of S[A]. But S[A] $\bar{c}$ R[A] after S<A=A]R is executed. Thus if we execute the semi-joins in sequence, their cost is at most 2 * size-of R[A], which is at most size-of R under the (reasonable) assumption that the "width" of A is less than or equal to the "width" of ATT_R-{A}. Given this assumption, the cost of the semi-joins is less than or equal to the cost of the join as claimed.

If we consider sequences of joins, however, the preceding analysis is not always valid; there are cases in which the composite execution of multiple joins is more cost beneficial than the corresponding semi-joins [BC]. However, we believe these cases to be uncommon. Furthermore, such cases are difficult to detect from statistical database characteristics, such as those of Figure 3.1, because the difference in effect between joins and semi-joins depends in a detailed way on the database state [BC]. Therefore, we choose to ignore join as a distributed query processing tactic.

3.2 Performance Estimation

Hereafter we will be concerned with constructing a reducer RHO for E whose reducing statements are drawn from OMEGA(E). Since every omega ∈ OMEGA(E) maps a reduction relative to E into another such reduction, every sequence of operations from OMEGA(E) also has this property; thus the logical correctness of RHO is guaranteed.

However, optimization considerations require that we construct a reducer that is efficient as well as correct. To do so we must estimate the performance of reduction operations. In particular, for each operation omega and database D, we need to estimate the effect, cost, and benefit of applying omega to D. Our techniques for this purpose are similar to those in [HY].

3.2.1 Profiles

To support these requirements, SDD-1 maintains a statistical description of the database, called a profile. Profiles contain the following information: For each relation R,

1. the number of tuples in R, denoted card(R);
2. the "width" of R, e.g. the number of bytes per tuple, denoted width(R) (we assume fixed-size tuples); and
3. for each attribute $A \in ATT_R$, the number of distinct values in R[A], denoted card(R[A]).

For each attribute A,

1. width(A) (we assume that A has the same width in each relation in which it appears); and
2. the number of distinct values in A's underlying domain, denoted card(dom(A)).

In using profiles, we assume that data values in each column of each relation are uniformly distributed over the tuples of the relation. We also assume all columns to be independent.

Profiles are updated off-line on a periodic basis to reduce overhead. The inaccuracies introduced by this time lag are acceptable because of the overall approximate nature of the process.

3.2.2 Effect Estimation

Let omega be an operation and $\bar{D}$ be a profile. The function effect(omega,$\bar{D}$) estimates the effect of omega on the database described by $\bar{D}$; its value is a profile $\bar{D}'$ that describes the (estimated) new state of that database.

If omega is a projection, e.g. R[X], effect(omega,$\bar{D}$) transforms width(R) into width(X) = $SUM_{A \in X}$width(A). In general, R[X] can also reduce the cardinality of R by collapsing previously distinct tuples into a single tuple. We do not attempt to estimate this effect except in two cases:

1. if X={A}, then card(R) is changed to card(R[A]);
2. if $PRODUCT_{A \in X}$(card(R[A])) < card(R), then card(R) is changed to equal that product.

A selection, e.g. R[A=k], affects card(R), and card(R[A']) for all $A' \in ATT_R$. Due to the uniformity assumption, the fraction of R tuples that satisfy the selection is

$$alpha_k = \begin{cases} 1 \,/\, card(R[A]), & \text{if } k \in R[A] \\ 0, & \text{otherwise} \end{cases}$$

and the expected cardinality of the result is $alpha_k$ * card(R). In practice it is prudent to assume k∈R[A]; if k∉R[A], the selection (and indeed the entire envelope) has a null result.

With this assumption, effect(R[A=k],$\bar{D}$) transforms card(R) into card(R)/card(R[A]), and card(R[A]) into 1. The effect on card(R[A']) for A' ≠ A is more complex and will be discussed momentarily.

The effect of a semi-join can be modelled as a sequence of selections. The fraction of R tuples expected to satisfy R<A=A]S is given by

$$SUM_{k \in S[A]} alpha_k$$

$$= SUM_{k \in S[A]} (1/card(R[A])) * (\text{the probability of } k \in R[A]).$$

Since we assume that columns are independent, the above probability is simply the probability that an arbitrary k∈dom(A) is also in R[A], which equals card(R[A])/card(dom(A)). Substituting into the above formulas yields

$$SUM_{k \in S[A]} (1/card(dom(A)))$$

$$= card(S[A]) / card(dom(A)).$$

Thus the estimated cardinality of the result is

card(R) * card(S[A]) / card(dom(A)).

The effect of R<A=A]S on card(R[A]) is estimated similarly to be card(R[A])*card(S[A])/card(dom(A)).

The effect of R<A=A]S, or equivalently R[A=k], on R[A'] for A'≠A is more complex. Given the independence assumption, we can analyze the effect as a hit ratio problem: we are given n=card(R) "objects", distributed uniformly over m=card(R[A']) "colors"; the question is, "How many colors are we expected to hit if we randomly select r of the objects?" where r is the expected cardinality of the resulting relation. The answer is given by [Yao]:

$$Y(m,n,r) = m * (1 - \text{PRODUCT}_{i=1}^{r}[(nd-i+1) / (n-i+1)],$$

where d = 1 - 1/m.

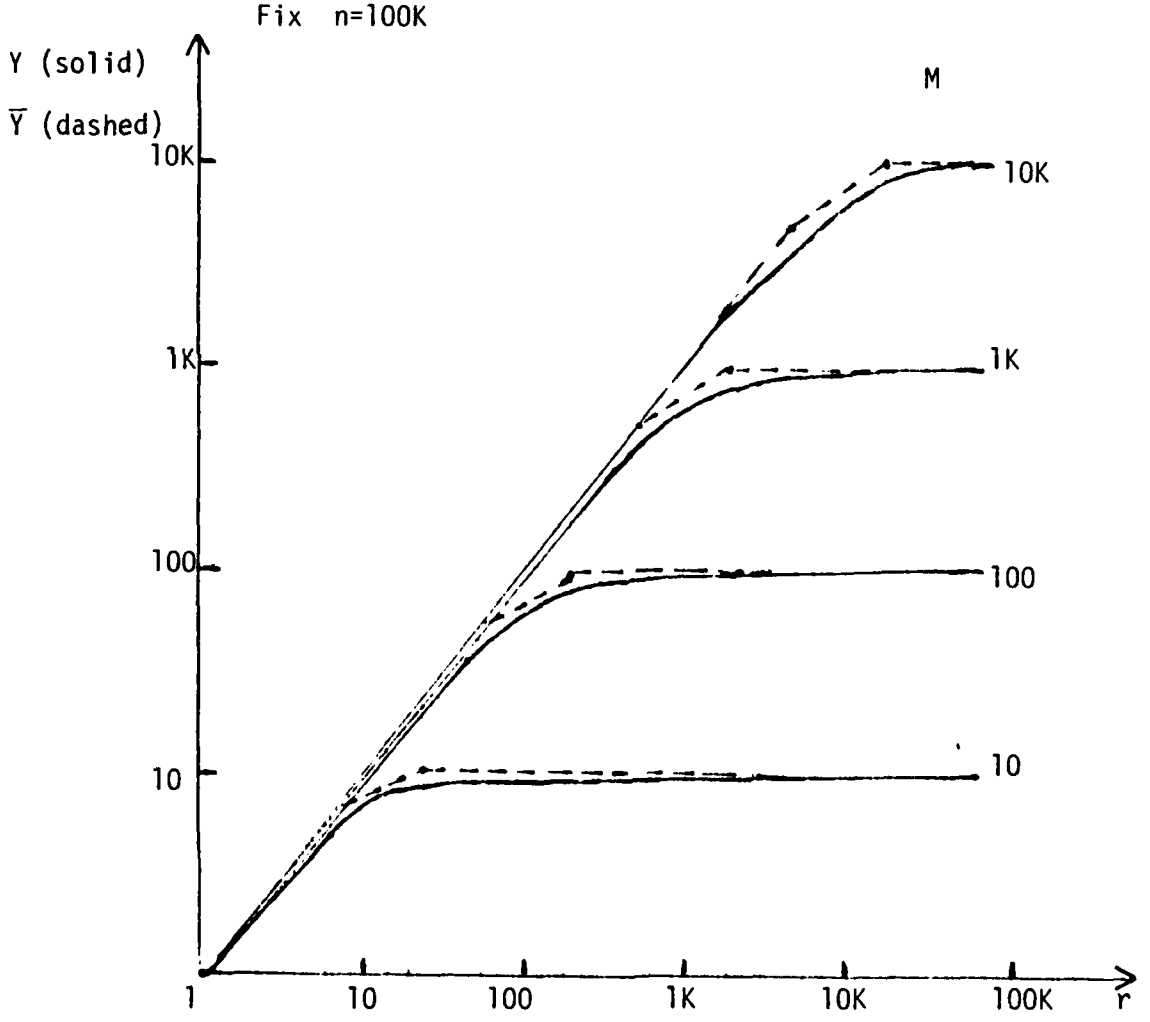
In practice, it is reasonable to approximate Y by

$$\bar{Y}(m,n,r) = \begin{cases} r & , \text{ for } r<m/2 \\ (r+m)/3 & , \text{ for } m/2 \leq r < 2m \\ m & , \text{ for } 2m \leq r \end{cases}$$

Y and $\bar{Y}$ are graphed in Figure 3.2.

The Yao Function, Y, and Approximation, $\bar{Y}$. Figure 3.2

Hit ratio problem: given n objects distributed over m colors;
question: how many colors Y will we hit if we select r objects?



3.2.3 Cost Estimation

cost(omega,$\bar{D}$) is defined to be 0 for all local operations, i.e. projections, selections, and semi-joins whose operands are stored at a single site. If omega is a non-local semi-join R<A=A]S, then

cost(omega,$\bar{D}$)=card(S[A])*width(A).

3.2.4 Benefit Estimation

Suppose omega reduces relation R; its benefit is defined to be the amount by which it reduces R, which equals the size of R minus the size of omega(R). Substituting results from 3.2.2, we get

benefit(R[X],$\bar{D}$)=width(R)-width(X), assuming card(R) is not also changed;

benefit(R[A=k],$\bar{D}$)=width(R)*(card(R)-card(R)/card(R[A]))
=width(R)*card(R)*(1-1/card(R[A]));

benefit(R<A=A]S,$\bar{D}$)=
width(R)*card(R)*(1-card(S[A])/card(dom(A))).

3.3 An Example Reducer

To illustrate the preceding material we now present an example reducer for envelope E_1 of Figure 2.3. The initial database profile is given in Figure 3.1. The reducer proceeds as follows.

1. SUPPLIER[STATE="MA"]

   effect: card(SUPPLIER) reduced to 5000/50=100

   card(SUPPLIER[STATE]) reduced to 1

   card(SUPPLIER[S#]) reduced to

   Y(5000,5000,100)=100

   cost: 0

   benefit: 65000-1300=63700

2. SUPPLY[S#,P#,PRICE]

   effect: width(SUPPLY) reduced to 3

   cost: 0

   benefit: 100000

3. PART[P#,FUNCTION,SPEED]

   effect: width(PART) reduced to 3

   cost: 0

   benefit: 30000

4. PART[FUNCTION="7401"]

effect: card(PART) reduced to 10000/200=50

card(PART[FUNCTION]) reduced to 1

card(PART[P#]) reduced to Y(10000,10000,50)=50

cost: 0

benefit: 30000-150=29850.

5. SUPPLY<P#=P#]PART

effect: card(SUPPLY) reduced to 100000*50/10000=500

card(SUPPLY[P#]) reduced to 5000*50/10000=25

card(SUPPLY[S#]) reduced to

Y(1000,100000,500)=500

cost: card(PART[P#])*width(P#)=50

benefit: 300000-1500=298500

6. SUPPLY<S#=S#]SUPPLIER

effect: card(SUPPLY) reduced to 500*100/5000=10

card(SUPPLY[S#]) reduced to 500*100/5000=10

card(SUPPLY[P#]) reduced to Y(25,500,10)=10

cost: card(SUPPLIER[S#])*width(S#)=100

benefit: 1500-30=1470

7. Assemble the reduction at site 1

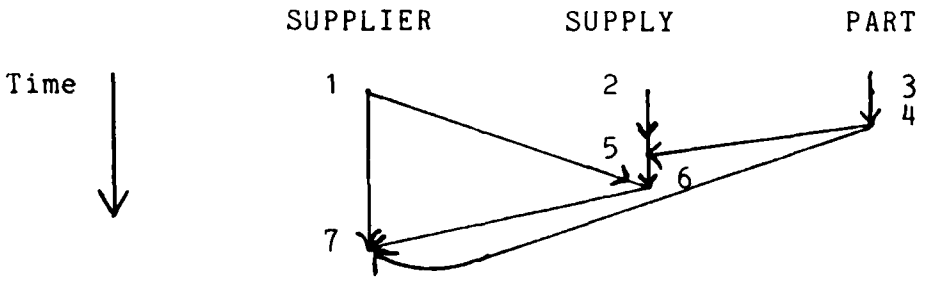
cost: card(SUPPLY)*width(SUPPLY)

+ card(PART)*width(PART)=30+150=180.

The total cost of this reducer is 330. By comparison, if no reducing operations were performed at all, it would cost 125000 to assemble the database at one site (site 2 in this case). And if only local operations were performed -- i.e. steps 1-4 -- the cost would be 1450.

The flow-graph of this reducer is shown in Figure 3.3; nodes in this graph correspond to operations, and arcs indicate data-flow between operations. Each operation can be executed as soon as all of its predecessors in the flow-graph have been executed, and thus substantial parallellism is possible. This parallellism is exploited by SDD-1 when it executes the reducer [RBFG].

Flow-Graph of Example Reducer Figure 3.3



numbers relate nodes in the graph to operations in the text.

---

4. Access Planning

The development of Sections 2 and 3 have mapped the original query optimization problem into the following more structured problem: we are given an envelope E; our goal is to construct a reducer RHO for E whose reducing statements are drawn from OMEGA(E), and whose cost is minimum over all such reducers. We call this problem access planning. This section presents our access planning algorithm. We emphasize that our solution is approximate, seeking to find low-cost, though not necessarily optimal, reducers. An algorithm that produces optimal reducers for a limited class of envelopes is presented in [HY]; no efficient algorithm for producing optimal reducers for general envelopes is known.

4.1 Algorithm AP

Our access planning algorithm is Algorithm AP, listed in Figure 4.1. Algorithm AP in an iterative optimization procedure whose main function is to construct a profitable sequence of reducing statements. In general terms AP

operates as follows. It initializes RHO to the null program and iteratively appends profitable, legal operations to RHO until all such operations have been used. Then the algorithm determines the cheapest site at which to assemble the reduction, and appends commands to move the reduction to that site. At this point RHO is the desired reducer, and the algorithm terminates.

We now examine AP in more detail.

Input/Output

The input to the algorithm is an envelope E and a database profile $\bar{D}$; its output is a reducer RHO for E whose reducing statements are estimated to be profitable when applied to the database described by $\bar{D}$.

State Space

The state of the algorithm at each iteration is determined by four variables.

1. RHO is the sequence of reducing statements constructed so far.
2. RHO($\bar{D}$) is the database profile that represents the estimated effect of applying RHO to the database described by $\bar{D}$; at each stage,

$$RHO(\bar{D}) = \underline{effect}(omega_k, (\underline{effect}(omega_{k-1}, (\ldots, (\underline{effect}(omega_1, \bar{D}))\ldots)))),$$

where $\langle omega_1, \ldots, omega_k \rangle = RHO$.

The Access Planner — Figure 4.1

Algorithm AP

Input: envelope E and database profile $\bar{D}$.
Output: RHO, a reducer for E.

State Space

RHO: a reducer for E.
RHO($\bar{D}$): database profile that results from executing RHO.
OMEGA: OMEGA(E)-{omega|omega is used in RHO}.
$OMEGA_{profitable}$: {omega$\in$OMEGA|benefit(omega,RHO($\bar{D}$)) $\geq$ cost(omega,RHO($\bar{D}$))}

Step 1 - Initialization

a. RHO:=null program.
b. RHO($\bar{D}$):=$\bar{D}$.
c. OMEGA:=OMEGA(E).
d. $OMEGA_{profitable}$:={omega$\in$OMEGA(E)|benefit(omega,$\bar{D}$) $\geq$ cost(omega,$\bar{D}$)}

Step 2 - Main Loop

a. Do while $OMEGA_{profitable} \neq \emptyset$
b. $omega_{best}$:=omega$\in$$OMEGA_{profitable}$ such that cost(omega,($\bar{D}$)) is minimum over all such omega; append $omega_{best}$ to RHO and remove from OMEGA and $OMEGA_{profitable}$.
c. RHO($\bar{D}$):=effect($omega_{best}$,RHO($\bar{D}$)); modify $OMEGA_{profitable}$ to reflect costs and benefits in new state (see text).
end

Step 3 - Termination

a. select assembly site:
   - for each site s,
     $cost_a$(s)=SUM, over all relations R stored at s, of width(R)*card(R);
   - the assembly site sa is the site s such that $cost_a$(s) is maximum over all sites.
b. append to RHO commands to move all relations to site sa.

END

3. OMEGA contains the legal operations not yet in RHO; these are the operations that can be added to RHO in future iterations. And
4. $OMEGA_{profitable}$ contains the operations that are estimated to be profitable in the state described by RHO($\bar{D}$).

Step 1 - Initialization

The four state variables are initialized to the appropriate values before the database has been reduced at all.

Step 2 - Main Loop

a. This step constructs a profitable sequence of reducing statements by repeating steps b & c until $OMEGA_{profitable}$ is exhausted.

b. On each iteration, the cheapest profitable operation, denoted $omega_{best}$, is appended to OMEGA.

An alternate approach would be to select the most profitable operation at each stage. However, suppose omega' has both high profit and high cost. Once we place omega' into RHO we have committed ourselves to paying its

high cost. But if we delay omega' and execute other less costly operations first, these operations may reduce the cost of omega' as a fringe benefit.

Notice that all local operations have zero cost and non-negative benefit, hence are always profitable and always have lower cost than any non-local operation. Consequently all legal local operations are placed into RHO before any non-local operations are. In practice the order of these local operations is important; but since this order is a matter of local query optimization it does not concern us here.

This step also removes $omega_{best}$ from $OMEGA_{profitable}$. There are, however, cases in which it is beneficial to re-use the same operation, possibly many times [BC]. We choose to ignore this possibility because such cases apparently arise infrequently, and it is difficult to bound the size of RHO otherwise.

c. RHO($\bar{D}$) is updated to reflect the estimated effect of $omega_{best}$, and $OMEGA_{profitable}$ is re-computed. To re-compute $OMEGA_{profitable}$, we need only check operations whose benefit or cost was changed by $omega_{best}$. In particular, suppose $omega_{best}$ is of the form R[X], R[A=k], or R<A=A]S. Then $omega_{best}$ reduces the size of R, and there are two further consequences:

1. the benefit of all other operations that reduce R is decreased; and
2. the cost of all semi-joins that use R to reduce another relation is also decreased.

Thus the operations that must be checked are:

1. {omega$\in$OMEGA$_{profitable}$ | omega reduces R}, and
2. {omega$\in$OMEGA - OMEGA$_{profitable}$ | omega = S'<A'=A']R, for any S',A'}.

Step 3 - Termination

a. Upon termination of Step (2), RHO is a program that computes a reduction for E. To complete its task, RHO must also assemble the reduction at a single site.

Let s1,...,sn be the sites housing data referenced by E, and let cost$_a$(si) be the sum, over all R at site si referenced by E, of width (R)*card(R). For any site sj, the cost of assembling the reduction at sj is

$$COST_a(sj)=SUM^{n}_{i=1,i\neq j}\ cost_a(si).$$

COST$_a$ is minimized by selecting the site with maximum cost$_a$ to be the assembly site.

b. Having selected the assembly site, the algorithm appends commands to RHO to move all relations to that site. At this point, RHO is a reducer for E, and Algorithm AP terminates.

4.2 Enhancements

Algorithm AP is an example of a greedy optimization algorithm; it always makes decisions on the basis of immediate gain, it never looks ahead, and it never backs up. As a result, the reducers generated by AP are in general sub-optimal. In this section we present two techniques for improving these reducers. Both techniques take a reducer RHO produced by the basic algorithm and transform it into a lower cost reducer RHO'.

The first enhancement operates by permuting the order of RHO to reduce the cost of some semi-joins without increasing the cost of any other operations. This technique is best understood in terms of flow graphs. Consider Figure 4.2a. Since the semi-join represented by arc (2) reduces S, the cost of semi-join (1) can be decreased by delaying it until after (2). The resulting reducer RHO' is shown in Figure 4.2b. Notice that the

reductive effect of semi-join (1) in RHO' is greater than its effect in RHO, because S[B] will be smaller in RHO', hence T<B=B]S will also be smaller. Consequently the cost of all semi-joins that follow (1) in the flow-graph are also reduced. Since no other semi-joins are affected by the transformation, RHO' is guaranteed to have lower cost than RHO.

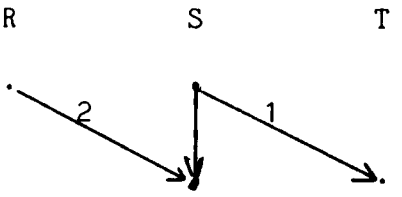
More generally, let (N_R, N_S) be any arc in RHO's flow graph going from the "R column" to the "S column" and let N_R' be any node in the R column after N_R. The replacement of (N_R, N_S) by (N_R', N_S) is guaranteed to monotonically decrease the cost of RHO, provided the resulting flow graph is acyclic. (If the resulting flow-graph contains a cycle, it no longer represents a physically executable program.)

To perform this transformation, we retain the values of RHO($\bar{D}$) computed at each step of the basic algorithm. When that algorithm terminates, we construct the flow graph of RHO and associate the retained values of RHO($\bar{D}$) with the corresponding nodes of the graph. Then we successively transform RHO by selecting the most expensive semi-join RHO and delaying it if possible: i.e. suppose (N_R, N_S) represents the most expensive semi-join in RHO, and let N_R' be the immediate successor of N_R in the R column, assuming N_R is not the last node in that column; we transform RHO

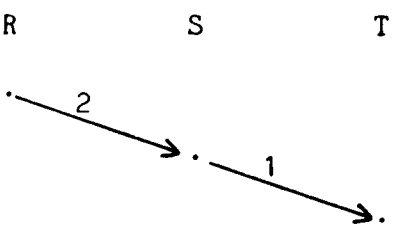
Improving RHO by Permuting Its Order Figure 4.2

qualification: R.A=S.A and S.B=T.B

(a) orginal reducer RHO



(b) better reducer RHO'



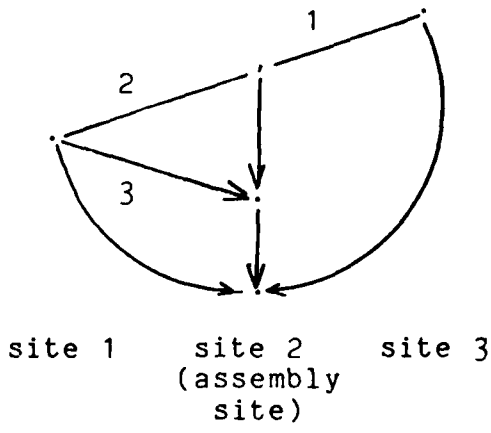
by replacing (N_R, N_S) by (N_R', N_S) provided the resulting graph is acyclic. Values of RHO($\bar{D}$) associated with N_S and its successors in the graph are updated to reflect the transformation and the process repeats until no more transformations are possible.

Our second enhancement seeks to prune operations from RHO that are rendered unprofitable by the choice of assembly site. Consider the reducer illustrated in Figure 4.3a, and suppose site 2 is the assembly site. Since relation S is stored at the assembly site, the semi-join S<A=A]R represented by arc (3) is superfluous and should be removed. The decision to incorporate this semi-join into

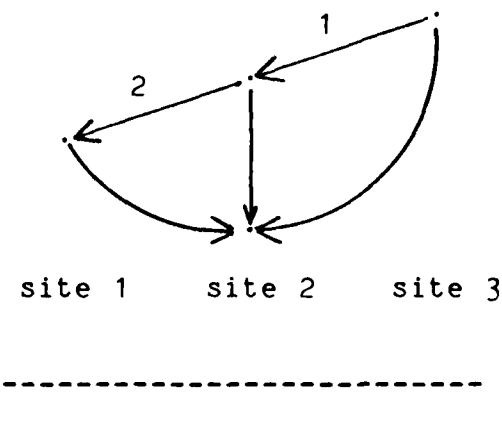
Improving by Pruning Semi-Joins Figure 4.3

qualification: R.A=S.A and S.B=T.B

(a) original reducer RHO



(b) transformed reducer RHO'



RHO was based on the belief that S would eventually be shipped to the assembly site; since S is already there, the benefit of the semi-join is zero. With respect to arc (1), i.e. S<A=A]T, the situation is less clear-cut: although there is no direct benefit in reducing S, this

semi-join is indirectly beneficial via arc (2); in fact, arc (1) both decreases the cost of arc (2), and increases its benefit!

In general, let (N_R,N_S) be any arc in the flow graph for RHO where S is a relation stored at the assembly site. The removal of (N_R,N_S) is beneficial if the cost of RHO minus (N_R,N_S) is less than the cost of RHO (including all assembly operations); the former cost is computed by removing (N_R,N_S) from the strategy graph and updating values of RHO($\bar{D}$) associated with N_S and its successors. We perform this test on all arcs of the form (N_R,N_S), considered in cost order.

The enhancements described in this section help compensate for the short-sightedness of Algorithm AP, by considering the indirect benefits of semi-joins. While these enhancements still fall short of optimality, they move in that direction.

5. Mapping Transactions Into Envelopes

Sections 3 and 4 have described a technique for efficient processing of envelopes. In this section we explain the transformation from Datalanguage transactions to envelopes. This transformation is described in two steps. Section 5.1 describes the mapping from transactions to logical envelopes, i.e. envelopes that reference logical relations; Section 5.2 then describes the mapping to envelopes that reference physical relations.

5.1 The Logical Transformation

The purpose of the logical transformation is to eliminate procedural aspects of Datalanguage transactions. Datalanguage is a rich language and a full treatment of this mapping is beyond the scope of this paper. Instead, we will describe the mapping for a representative subset of the language.

5.1.1 A Subset of Datalanguage

A Datalanguage transaction is either a statement or a sequence of statements bracketed by Begin ... End. We will consider four types of statements: iteration-statements, conditional-statements, assignment-statements, and print-statements; Datalanguage does not include goto.

Two types of variables exist in Datalanguage: indexed-variables which represent database values (see below), and program-variables which are global variables in the style, say, of FORTRAN. Referring to transaction T_2 of Figure 5.1, PART.P# and PART.SPEED are indexed-variables, while COUNT is a program-variable.

Assignment and print-statements are self-explanatory and will not be described in detail. Conditional-statements have the form

If boolean then body1 [else body2];

where body1 and body2 are either statements or sequences of statements bracketed by Begin ... End. Conditional-statements are interpreted with the usual semantics.

Transaction T_2 Figure 5.1

```
Begin
  COUNT:=0;
  For PART
     If PART.SPEED=2
     Then Begin
          If COUNT < 50
          Then Begin
            Print PART.P#;
            COUNT:=COUNT+1;
          End;;;
End
```

Iteration-statements have the form

For relation body1;

where body1 is defined above. This statement is interpreted as follows. body1 is executed once per tuple of relation with each indexed-variable of the form relation.attribute instantiated by the value of attribute in that tuple. For example, the statement

```
For PARTS
   If PARTS.FUNCTION=7401
   Then Print "Part Number =",PARTS.P#;;
```

prints the P# of every tuple in PARTS with FUNCTON=7401. We assume that each iteration-statement in a transaction references a different relation; if two iteration-statements reference the same relation a construct similar to "tuple variables" is employed.

5.1.2 Transformation to Logical Envelope

Given a transaction T, we obtain the target-list of its envelope by listing all indexed-variables that appear in T.

To obtain the qualification for T's envelope, the following recursive definition is applied. Let body=<statement$_1$,...,statement$_n$>. We define

$$\text{Qual}(\underline{\text{body}}) = \text{OR}_{i=1}^{n}\ \text{Qual}(\underline{\text{statement}}_i), \text{ where}$$

$$\text{Qual}(\underline{\text{statement}}_i) = \begin{cases} \text{(a) True, for } \underline{\text{assignment}} \text{ and } \underline{\text{print statements}}; \\ \text{(b) } (\underline{\text{boolean}} \text{ AND Qual}(\underline{\text{body1}})) \\ \quad [\text{OR (NOT } \underline{\text{boolean}} \text{ AND Qual}(\underline{\text{body2}}))], \\ \quad \text{for } \underline{\text{conditional statements}}; \\ \text{(c) Qual}(\underline{\text{body1}}), \\ \quad \text{for } \underline{\text{iteration statements}}. \end{cases}$$

The qualification for T's envelope is obtained by (1) removing all non-iteration-statements from T, yielding T', (2) constructing Qual(T'), and (3) replacing every clause that contains a program-variable by True. This procedure is illustrated in Figure 5.2.

5.1.2 Transformation to Logical Envelope

Given a transaction T, we obtain the target-list of its envelope by listing all indexed-variables that appear in T.

To obtain the qualification for T's envelope, the following recursive definition is applied. Let body=<statement_1,...,statement_n>. We define

$$\text{Qual(body)} = \text{OR}_{i=1}^{n}\ \text{Qual}(\text{statement}_i), \text{ where}$$

$$\text{Qual}(\text{statement}_i) = \begin{cases} \text{(a) True, for assignment and print statements;} \\ \text{(b) (boolean AND Qual(body1)) [OR (NOT boolean AND Qual(body2))], for conditional statements;} \\ \text{(c) Qual(body1), for iteration statements.} \end{cases}$$

The qualification for T's envelope is obtained by (1) removing all non-iteration-statements from T, yielding T', (2) constructing Qual(T'), and (3) replacing every clause that contains a program-variable by True. This procedure is illustrated in Figure 5.2.

It is easily proved that this transformation is correct, i.e. for all T it yields an envelope E such that T(E(D))=T(D) for all states of D. Moreover, if T contains no program-variables the transformation is exact, i.e. for all states of D, E(D) is the minimum state such that T(E(D))=T(D). Various code optimization techniques could be employed to improve the logical transformation when program-variables are present. However, this issue is not addressed here.

5.2 The Physical Transformation

At the physical level, SDD-1 permits each logical relation to be partitioned into sub-relations called fragments, which are the units of data distribution. Each fragment may be stored at one or more sites; each stored instance of a fragment is called a stored fragment. The relationship between relations, fragments, and stored fragments is illustrated in Figure 5.3. The purpose of the physical transformation is to map an envelope E that references logical relations into an envelope E_{SF} that references stored fragements.

The fragments of a relation are defined in two steps. First, the relation is partitioned "horizontally" into

Logical Transformation — Figure 5.2

(1) Remove non-iteration-statements

T_2': < For PART
If PART.SPEED = 2
Then If COUNT < 50
Then Begin
Print PART.P#
COUNT:=COUNT+1;
END;;;>

(2) Construct Qual(T_2')

= Qual(For PART ...;)

= Qual(If PART.SPEED = 2 Then ...;)

= PART.SPEED = 2 AND Qual(If COUNT < 50 Then ...;)

= PART.SPEED = 2 AND COUNT < 50 AND Qual(Print PART.P#;
COUNT := COUNT + 1;)

= PART.SPEED = 2 AND COUNT < 50 AND (True OR True)

(3) Replace clauses that contain program-variables by True.
In this case, replace COUNT < 50 by True

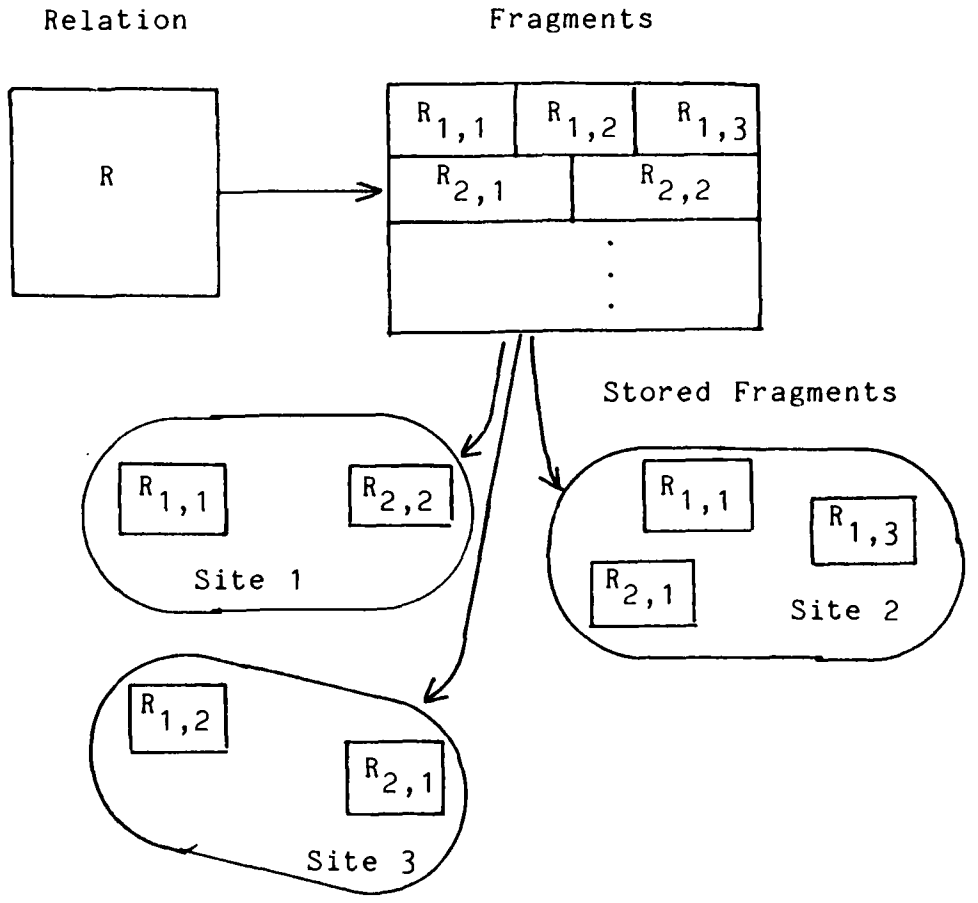
The resulting qualification is

PART.SPEED = 2

---

subsets defined by selection formulas (see Figure 5.4), and then each horizontal subset is partitioned vertically into sub-relations defined by projections (see Figure 5.5). In addition, a unique tuple identifier (abbr. TID) is appended to each tuple and included in every fragment

Relationship Between Relations and Stored Fragments — Figure 5.3



to guarantee lossless reconstruction of the original relation [DB]. We use $R_1,\ldots,R_n$ to denote the horizontal subsets of relation R, and $R_{i,1},\ldots,R_{i,m}$ to denote the vertical subrelations of R_i; notice that for all states of the database

$$R_i = R_{i,1}[TID=TID]R_{i,2}[TID=TID] \ldots [TID=TID]R_{i,m}$$

and $R = R_1 \cup R_2 \cup \ldots \cup R_n$.

Horizontal Fragmentation Figure 5.4

Relation
PART(P#, FUNCTION, SPEED, PACKAGE)

Horizontal Fragments
$PART_1$=PART[SPEED=1]
$PART_2^1$=PART[SPEED=2 and PACKAGE="16 pin DIP"]
$PART_3^2$=PART[SPEED=2 and PACKAGE≠"16 pin DIP"]
$PART_4^3$=PART[SPEED>2]

Vertical Fragmentation Figure 5.5

1. Horizontal Fragment
   $PART_1$(P#, FUNCTION, SPEED, PACKAGE)

   Vertical Fragments
   $PART_{1,1}$=$PART_1$[P#, FUNCTION]
   $PART_{1,2}^{1,1}$=$PART_1$[SPEED]
   $PART_{1,3}^{1,2}$=$PART_1$[PACKAGE]

2. Horizontal Fragment
   $PART_2$(P#, FUNCTION, SPEED, PACKAGE)

   Vertical Fragments
   $PART_{2,1}$=$PART_2$[P#, SPEED]
   $PART_{2,2}^{2,1}$=$PART_2$[FUNCTION, PACKAGE]

3. etc.

Given an envelope E (referencing logical relations) we obtain an envelope E_F that references fragments by applying a query modification procedure described in [Dayal,DB]. This procedure maps each clause of the form R.A=S.A into a formula $OR_{i=1}^{nr}(OR_{j=1}^{ns}(R_i.A=S_j.A))$, where $R_1,\ldots,R_{nr}$ and $S_1,\ldots,S_{ns}$ are the horizontal fragments of

R and S respectively[7]. Then each indexed-variable $R_i.A$ is replaced by $R_{i,k}.A$, where $R_{i,k}$ is the vertical fragment of R_i that includes attribute A. (Since the vertical fragments partition R_i, the choice of $R_{i,k}$ is unique.) Finally, the vertical fragments of each horizontal fragment are "joined" on TID, by appending the formula $AND_{k=1}^{mr-1}(R_{i,k}.TID=R_{i,k+1}.TID)$ to the qualification, where $R_{i,1},\ldots,R_{i,mr}$ are the vertical fragments of R_i referenced by the envelope. The result is the qualification of E_F; the target-list is obtained similarly.

E_F is then "improved" by detecting and discarding horizontal fragments whose definitions contradict the qualification. To do so, E_F is placed into disjunctive normal form and for each conjunct C the following test is performed. We append to C the selection formulas that define each horizontal fragment referenced in C. Then we test the satisfiability of the resulting formula using mechanical theorem-proving techniques. If the formula is unsatisfiable, C is removed from the qualification.

Given E_F, the remaining task is to obtain an envelope E_{SF} that references stored fragments. In principle, this

---

7. Selection clauses of the form R.A=k are mapped into $OR_{i=1}^{nr} R_i.A=k$.

transformation entails an optimization problem. It is, however, an optimization problem we choose not to address in SDD-1. Instead the mapping is accomplished via table look-up: for each site there is a pre-defined table, called a materialization, which specifies the stored instance of each fragment to use in processing transactions submitted at that site.

At this point, E_{SF} is an envelope in the form assumed by sections 2-4, and query processing proceeds as described there.

6. Conclusion

We have described query processing in SDD-1 as a three step procedure in which (1) a transaction is transformed into an envelope, (2) the envelope is compiled into a program called a reducer that assembles a reduction of the database at a single site, and (3) the transaction is executed against the reduction at that site. This approach separates issues that are transaction-language specific (steps (1) and 3)) from those of distributed query optimization (step (2)). This paper has concentrated on the latter issue; the optimization techniques presented in this paper are usable in other distributed relational DBMSs, as long as the translation from transactions to envelopes is feasible.

Our treatment has left many problems open. Among the most pressing are

1. finding ways of helping the optimization algorithm avoid entrapment by high-cost local optima;
2. use of feedback to compensate for inaccuracies in performance estimation; and

3. dynamic selection of stored fragments, e.g. to maximize the clustering of accessed fragments at individual sites.

We also believe that our methods can be extended to non-relational systems. Recent work by [Dayal, Zaniolo] on building relational interfaces to CODASYL databases suggest that our optimization techniques can be adapted to this setting. However, this too remains a matter for future research.

References

[AHU]
Aho, A.V., J. Hopcroft, and J.D. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley, Reading, MA, 1974.

[BC]
Bernstein, P.A., and D.W. Chiu, "Using Semi-Joins to Solve Relational Queries," to appear JACM.

[BG]
Bernstein, P.A., and N. Goodman, "Full Reducers for Relational Queries Using Multi-Attribute Semi-Joins," Proc. 1979 NBS Symp. on Comp. Netw., Dec. 1979.

[BSR]
Bernstein, P.A., D.W. Shipman, and J.B. Rothnie, "Concurrency Control in SDD-1: A System for Distributed Databases", to appear ACM TODS.

[CCA]
Datacomputer Users Manual, Comp. Corp. of Am., Cambridge, MA, July, 1978.

[Dayal]
Dayal, U. Schema Mapping Problems in Database Systems, Tech. Rep. TR-11-79, Center for Research in Computing Technology, Harvard University, August, 1979.

[DB]
Dayal, U., and P.A. Bernstein, The Fragmentation Problem: Lossless Decomposition of Relations into Files, Tech. Rep. CCA-78-13, Comp. Corp. of Am., Cambridge, MA, November 1978.

[ESW]
Epstein, R., M. Stonebraker, and E. Wong, "Distributed Query Processing is a Relational Database System", Proc. 1978 Berkeley Work. on Dist. Data Management and Comp. Netw., May 1978

[HS]
Hammer, M.M., and D.W. Shipman, Reliability Mechanisms for SDD-1: A System for Distributed

Databases, Tech. Rep. CCA-79-05, Comp. Corp. of Am., Cambridge MA, July 1979.

[HSW]
Held, G., M. Stonebraker, and E. Wong, "INGRES: A Relational Database System", Proc. 1975 NCC, AFIPS Press, Montvale NJ.

[HY]
Hevner, A.R., and S.B. Yao, "Query Processing in Distributed Databases", IEEE Trans. on Soft. Eng., Vol SE-5, No. 3, May 1979.

[Rothnie]
Rothnie, J.B., "Evaluating Inter-Entry Retrieval Expressions in a Database Management System", Proc. 1975 NCC, AFIPS Press, Montvale NJ.

[RBFG]
Rothnie, J.B., P.A. Bernstein, S.A. Fox, N. Goodman, M.M. Hammer, T.A. Landers, D.W. Shipman, C.L. Reeve, and E. Wong, "SDD-1: A System for Distributed Databases", to apper ACM TODS.

[RGM]
Rothnie, J.B., N. Goodman, and T. Marill, "Database Architecture in a Network Environment", in Protocols and Techniques for Data Communication Networks, F.F. Kuo ed., Prentice-Hall, 1978.

[Willcox]
Willcox, D.A., "Optimization of a Relational Algebra Query to a Distributed Database Using Statistical Sampling Methods", Doc #234, Center for Advanced Computation, Univ. of Ill., August, 1977.

[Wong]
Wong, E., "Retrieving Dispersed Data in SDD-1: A System for Distributed Databases", Proc. 1977 Berkeley Workshop on Dist. Data Man. and Comp. Netw., May 1977.

[Yao]
Yao, S.B., "Approximating Block Accesses in Database Organizations", CACM, Vol. 20, No. 4, April 1977.

[Zaniolo]
Zaniolo, C., "Design of Relational Views Over

Network Schemes", Proc. 1979 ACM SIGMOD Conference, June 1979.